SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Preddiplomski sveučilišni studij računarstva

Završni rad

IMPLEMENTACIJA ČESTIČNOG MODELA

IMPLEMENTATION OF PARTICLE SYSTEMS MODEL

Rijeka, rujan 2015.

Enzo Licul

0069058647

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Preddiplomski sveučilišni studij računarstva

Završni rad

IMPLEMENTACIJA ČESTIČNOG MODELA

IMPLEMENTATION OF PARTICLE SYSTEMS MODEL

Mentor: doc. dr. sc. Jerko Škifić

Rijeka, rujan 2015.

Enzo Licul

0069058647

Izjava o samostalnoj izradi zadatka

Ovim dokumentom izjavljujem da sam završni rad izradio uz pomoć smjernica i uputa mentora, dr.sc. Jerka Škifića, te samostalnim istraživanjem i radom.

Enzo Licul

Potpis: _____

Zahvaljujem mentoru doc. dr. sc. Jerku Škifiću koji mi je omogućio sudjelovanje na projektu. Također zahvaljujem na usmjeravanju i uvijek dobrodošloj potpori.

Posebno se želim zahvaliti svojim roditeljima i cijeloj obitelji na strpljenju, poticanju i razumijevanju.

Hvala svim kolegama i prijateljima koji su doprinijeli moralnoj podršci prilikom istraživanja i izrade završnog rada.

Sadržaj

1. Uvod

1.1. Opis sadržaja zadatka

Ovaj rad opisuje princip izrade računalnog programa za simulaciju gibanja čestica u dvodimenzionalnom prostoru. U radu su modelirane različite vrste čestica, implementirane su globalne sile odbijanja i privlačenja te je primijenjena tekstura na sustav čestica.

Projekt je započet s ciljem učenja OpenGL grafike i fizičke knjižnice Box2D. Kao rezultat razvijena je aplikacija koja simulira dvodimenzionalni sustav čestica sa mogućnošću djelovanja fizike na njih.

1.2. Računalna grafika

Računalna grafika je područje računarstva koje se bavi proučavanjem metoda za digitalnu sintezu i manipulaciju vizualnog sadržaja. Sam pojam računalna grafika obuhvaća proučavanje trodimenzionalne i dvodimenzionalne grafike i obradu slika. Danas je računalna grafika jako popularna jer nam znatno olakšava i ubrzava rad na računalima. Primjer je korištenje GUI-a (graphical user interface) umjesto klasičnog TUI-a (text user interface) kako ne bismo trebali sami ručno upisivati komande.Današnju računalnu grafiku pretežito koristimo za: zabavu (produkcije filmove, filmski efekti, video igrice), znanost i inženjerstvo (projektiranje uz pomoć računala, vizualizacija), virtualne prototipe, treninge i simulacije i za grafičku umjetnost. Postoje dvije vrste računalne grafike:

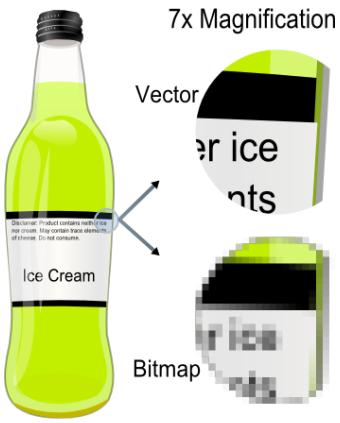
a) rasterska grafika – gdje je svaki piksel posebno definiran.

b) vektorska grafika – gdje se upotrebljavaju matematičke formule za crtanje linija i oblika. [1]

Upotrebom vektora dobiju se oštre grafike i često manji računalni dokumenti, ali kada su vektorske grafike komplicirane, tada je potrebno puno više vremena da se učitaju i mogu imati veće računalne dokumente nego rasterski ekvivalenti. Na slici 1.1 moguće je vidjeti razliku između vektorske i rasterske grafike.

Od samih početaka kad se računalna grafika bazirala uglavnom na dvodimenzionalnoj grafici do danas kad se uglavnom koristi za trodimenzionalnu grafiku računalna grafika se znatno poboljšala, pa današnje slike izgledaju znatno realističnije.

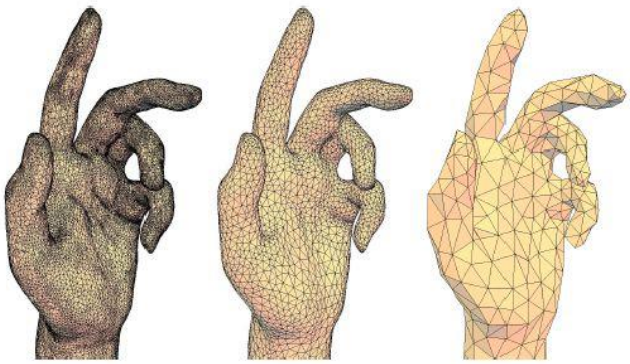
Računalnu grafiku možemo podijeliti na puno grana, ali neke od najbitnijih grana računalne grafike jesu: geometrija (eng. geometry), animacija (eng. animation), renderiranje (eng. rendering), obrada slike (eng. imaging) i topologija (eng. topology). Sljedeća poglavlja ukratko opisuju svako pojedino područje računalne grafike.



Slika 1.1: Razlika između vektorske i rasterskog prikaza grafike. [2]

1.2.1. Geometrija

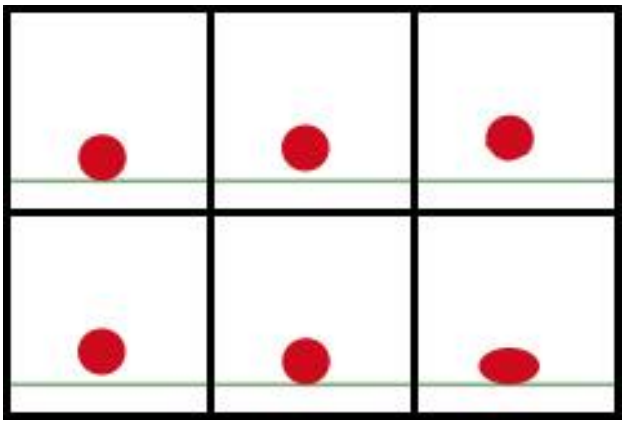
Područje geometrije bavi se proučavanjem reprezentacije objekata u tri dimenzije. Budući da izgled objekta znatno ovisi o njegovoj vanjskoj površini, koristimo različite diskretne digitalne aproksimacije kako bi se naša površina približila stvarnom izgledu objekta. Najčešća reprezentacija površine se bazira na reprezentaciji putem poligonalna mreže (eng. polygonal meshes), iako je točkom-bazirana (eng. point-based) reprezentacija postala sve popularnija u zadnje vrijeme. Takve se reprezentacije zasnivaju na matematičkim metodama poput Lagrangianove, Eulerove i mnogih drugih.



Slika 1.2: Prikaz geometrijske mreže modela s različitim brojem točaka. [3]

1.2.2. Animacija

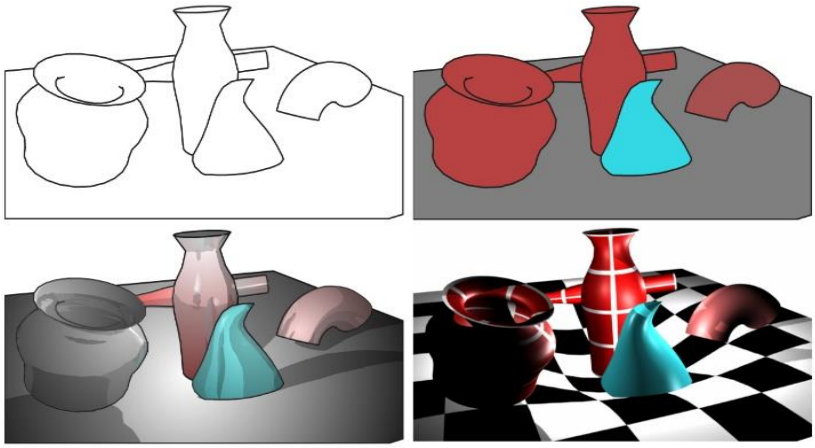
Računalna animacija nam omogućava stvaranje pokretnih slika upotrebom računala. Ona je potkategorija računalne grafike i animacije. Ponekad animaciju koristimo isključivo na računalu, a ponekad i u drugim medijima kao što su filmovi. Za stvaranje iluzije pokreta potrebno je najprije stvoriti sliku na zaslonu, a zatim je brzo zamijeniti slikom koja je slična prethodnoj slici, ali neznatno pomaknuta. Ovakva je tehnika identična tehnici iluzije pokreta na televizijama i pokretnim slikama. [4]



Slika 1.3: Prikaz slika koje kombinirane zajedno stvaraju animaciju loptice.

1.2.3. Renderiranje

Renderiranje je postupak dodavanja boje, teksture, osvjetljenja na osnovni dvodimenzionalni ili trodimenzionalni model koji je do tada sačinjen samo od linije i vrhova s ciljem stvaranja realističnih scena. U trodimenzionalnoj grafici postoje dvije vrste renderiranja, a to su sporo pre-renderiranje i renderiranje u stvarnom vremenu.Pre-renderiranje je intenzivan proces, a većinom se koristi za stvaranje filmova. Renderiranje u stvarnom vremenu se najčešće koristi kod video igrica, gdje se grafička scena obrađuje u stvarnom vremenu i prikazuje na monitoru. Tijekom cijelog procesa trodimenzionalnog renderiranja broj refleksija "svjetlosnih zraka" može biti promijenjen da bi postigli željeni grafički efekt.



Slika 1.4: Prikaz različitih postupaka renderiranja scene. [5]

1.2.4. Obrada slike

Obrada slike je postupak promijene odrađenog broja detalja na nekoj digitalnoj fotografiji. Postoje mnogi postupci mijenjanja digitalne fotografije, a neki od osnovnih postupaka jesu: otklanjanje nečistoća, uklanjanje neželjenih dijelova, promjena određene boje, položaj slike, dodavanje efekata i mijenjanje dubine boje.

Slika 1.5: Prikaz uklanjanja neželjenih dijelova sa digitalne slike. [6] [7]

1.2.5. Topologija

Topologija može biti formalno definirana kao učenje kvalitativnih svojstva pojedinih objekata koji su nepromijenjeni pod određenom vrstom transformacije.U topologiji je važnije kvalitativno nego kvantitativno. Bitno je reći da se topologija ne temelji na pitanju kako podaci izgledaju, već kako su oni povezani. Matematičke pozadine koje služe za stvaranje oblika računalne grafike se temelje prvenstveno na topologiji.

2. Opis korištenih alata i programskog jezika

Prilikom izrade ovog projekta korišteni su različiti alati. Za pisanje koda aplikacije korištena ja razvojna okolina QtCreator, a kod je pisan u programskom jeziku C++ na operacijskom sustavu Linux (Kubuntu 14.10). Kako bi se izbjegli mogući problemi oko kompajliranja, buildanja i pokretanja programa korišten je program CMake. Za vizualizaciju pojedinih čestica korištena je grafička knjižnica OpenGL, a pošto nam OpenGL ne pruža mogućnost rada s prozorima operacijskog sustava korištena je multimedijska knjižnica SDL. SDL ćemo koristiti i prilikom učitavanja tekstura i upravljanjem s događajima koji nastaju prilikom pritiska odrađene tipke na tipkovnici ili mišu. Za dodavanje fizike na sustav čestica korištena je fizička knjižnica Box2D, a GUI je stvoren uz pomoć male i jednostavne C++ knjižnice pod nazivom AntTweakBar.

2.1. Programski jezik C++

Programski jezik C++ jedan je od najpopularnijih programskih jezika koji se koristi za opću namjenu, a ima podršku za objektno orijentirano programiranje. Ovaj jezik je vrlo složen i velikih mogućnosti, a koristi se na velikim i osobnim računalima. Upravo razlozi kao što su djelotvornost, prilagodljivost i dobra normiranost programskog jezika C++ omogućili su ovoliko veliku popularnost jezika. Početci razvijanja ovog jezika započeli su u Bell Labs-u pod vodstvom danskog znanstvenika Bjarnea Stroustrupa tokom 1980-tih i to kao proširenje programskom jeziku C, pa mu je originalno ime bilo "C sa klasama" (eng. "C with classes"). Velika potražnja za objektno orijentiranim jezicima dovela je 1998. godine do ratifikacije programskog jezika C++ kao ISO/IEC 14882 standarda. Treba napomenuti kako jezik C++ nije u ničijem vlasništvu za razliku od jezika poput Jave koja je u vlasništvu Oracle-a. Daljnji razvoj programskog jezika C++ je pod nadzorom Međunarodnog odbora za standardizaciju. [8]

2.2. CMake

CMake je besplatni cross-platformni program otvorenog koda (eng. open source) koji nam služi za upravljanje procesom kompajliranja pojedinih projekata. Dizajniran je s podrškom prema aplikacijama koje koriste više knjižnica, a pri kompajliranju koristi jednostavne platformski-neovisne i kompajlerski-neovisne konfiguracijske datoteke. Proces izgradnje

projekta s CMake-om se odvija u dvije faze. Prvo se kreiraju standardne "build" datoteke koje su stvorene iz konfiguracijske datoteke. Zatim koristimo izvršne platformske alate pri izgradnji izvršnih datoteka. Svaki "build" projekt u svom direktoriju sadrži CMakeLists.txt datoteku koja kontrolira proces izgradnje projekta. [9]

CMake koristi datoteku CMakeLists.txt za kreiranje projekata. Uz pomoć ove datoteke možemo kontrolirati direktorije, datoteke i knjižnice koje želimo da budu uključene u projektu.

```
	cmake_minimum_required(VERSION 2.6)
	PROJECT (particles)

	set(BOX2D_VERSION 2.3.0)

	add_subdirectory(Box2D)

	file(COPY ${CMAKE_SOURCE_DIR}/data DESTINATION
${CMAKE_SOURCE_DIR}/build/)

	find_package(OpenGL REQUIRED)
	find_package(SDL2 REQUIRED)
	find_package(SDL2_image REQUIRED)

	include_directories (
		${OPENGL_INCLUDE_DIR}
		${SDL2_INCLUDE_DIR}
		${SDLIMAGE2_INCLUDE_DIR}
	)

	LINK_LIBRARIES(${SDL2_LIBRARY} ${SDLIMAGE2_LIBRARY}
${SDL2MAIN_LIBRARY})

	add_library( ANTTWEAKBAR_SDL STATIC
		${ANTTWEAKBAR_SOURCE}
		${ANTTWEAKBAR_HEADERS}
	)

	ADD_EXECUTABLE(main particles/main.cpp)

	target_link_libraries (
		main
		Box2D
		ANTTWEAKBAR_SDL
		${OPENGL_LIBRARIES}
		${SDL2_LIBRARIES}
		${SDL2IMAGE_LIBRARY}
		${SDL2MAIN_LIBRARY}
	)
```

Kod 2.1: Prikaz koda datoteke CMakeLists.txt.

Na kodu 2.1 možemo vidjeti prikaz naredba koje CMake sadrži te sam izgled CMakeLists.txt konfiguracijske datoteke. Naredbe prve i druge linije koda služe za definiranje minimalne verzije CMake-a i za dodjeljivanje naziva projektu. Prilikom dodjeljivanja određene

vrijednosti željenoj varijabli koristimo naredbu `set()`. Naredbom `add_subdirectory()` dodajemo poddirektorij koji u sebi sadrži CMakeLists.txt konfiguracijsku datoteku. Knjižnice koje su instalirane na našem operacijskom sustavu pronaći ćemo uz pomoć naredbe `find_package()`. Ukoliko želimo dodati neku knjižnicu koristit ćemo se naredbom `add_library()`, a povezivanje različitih knjižnica omogućit će nam naredba `LINK_LIBRARIES()`. Za povezivanje svih potrebnih knjižnica koje će program koristiti koristimo naredbu `target_link_libraries()`, a naredba `ADD_EXECUTABLE()` koristi se prilikom stvaranja programa za pokretanje.

2.3. OpenGL

OpenGL (eng. Open Graphics Library) je grafička knjižnica koja nam omogućava stvaranje animacija, visokokvalitetne dvodimenzionalne i trodimenzionalne računalne grafike. Izvorno je stvorena od strane Silicon Graphics Inc (SGI) tokom osamdesetih i devedesetih godina prošlog stoljeća, kako bi se pojednostavio potreban kod za pisanje dvodimenzionalnih i trodimenzionalnih aplikacija. Podržava više različitih programskih jezika i radi na više različitih platforma. Sučelje OpenGL-a čini više od 250 različitih funkcija koje se koriste prilikom crtanja složenih trodimenzionalnih objekata iz osnovnih geometrijskih oblika. Mnoge su mogućnosti koje nam OpenGL pruža, kao na primjer primjena transformacije nad objektima u sceni, različite vrste projekcije, odbacivanje promatraču ne vidljivih dijelova objekata, primjena tekstura na tijelo i mnoge druge. Najrasprostranjeniji je u CAD (eng. Computer-Aided Design) sustavima, GIS (eng. Geographic Information System) sustavima, u sustavima za stvaranje virtualne stvarnosti, simulacije leta, znanstvene vizualizacije i u izradi video igara. [10]

Iako je najnovija trenutna verzija OpenGL-a verzija 4.5 u ovom je projektu korištena verzija 2.0, uglavnom zbog toga jer je praktički kompatibilna sa svim grafičkim karticama i pomalo jednostavnija za razumijevanje za razliku od novijih modela koji koriste 'pipeline' arhitekturu. Kod pisanja kod-a pazilo se na code reuse odnosno sav OpenGL kod pisan je u posebnoj utils.cpp datoteci koja se kasnije u glavnom (main.cpp) programu poziva pomoću utils.h datoteke. To je zbog toga kako ne bismo imali duplicirani kod prilikom stvaranja svakog novog primjera različitih sustava čestica i kako bi nam kod bio pregledniji.

```
void init()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0,WIDTH,HEIGHT,0,-1,1);
    glMatrixMode(GL_MODELVIEW);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    world = new b2World(b2Vec2(0.0, 9.81));
}
```

Kod 2.2: Prikaz init() funkcije koja se nalazi unutar utils.cpp datoteke.

Na prikazanom kodu možemo vidjeti osnovne funkcije postavki OpenGL za stvaranje scene s crnom bojom. Početna funkcija `glMatrixMode(GL_PROJECTION)` nam služi za određivanje matrice koja će biti korištena i u koji stog će biti spremljena kao početna matrica. Pozivom funkcije `glLoadIdentity()` stvorit ćemo novu jediničnu matricu identiteta, a funkcija `glOrtho()` će nam poslužiti pri skaliranju OpenGL koordinata prema unesenim veličinama piksela (`WIDTH, HEIGHT`). Posljednja OpenGL funkcija unutar ovog koda je `glClearColor()` koja nam omogućava postavljanje pozadinske boje. Funkcija `b2World()` pripada Box2D knjižnici i bit će objašnjena u Box2D poglavlju. Funkciju `init()` pozivamo samo jednom prilikom pokretanja programa.

U nastavku možemo vidjeti jedan isječak koda koji se nalazi unutar `display()` funkcije u utils.cpp datoteci. Kao što i sam naziv govori funkcijom `glClear()` brišemo odabrane međuspremnike (.eng buffer) memorije. Preostale funkcije koristimo kako bi omogućili alpha blending odnosno dodavanje i omogućavanje prozirnosti naših boja. Treba napomenuti kako se `display()` funkcija poziva svaki puta kada je nešto potrebno nacrtati, što praktički znači da se ova funkcija poziva stalno dok se naš program izvršava.

```
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Kod 2.3: Prikaz isječka display() funkcije koji sadrži OpenGL funkcije.

Na kodu 2.4 možemo vidjeti implementaciju funkcije `drawParticle()` koja nam služi za crtanje pojedinih čestica. Ovu funkciju pozivamo i koristimo ukoliko ne želimo koristit nekakve teksture na sustav čestica, već želimo da naše čestice sadrže boju koju smo im unijeli uz pomoć GUI-a. Funkcijom `glColor4f()` određujemo kojom će se bojom crtati naše čestice. Prilikom svake izmjene matricu moramo prenijeti na stog naredbom `glPushMatrix()` a zatim na njoj vršimo radnje translacije (`glTranslatef()`) i rotacije

(`glRotatef()`). Za crtanje čestice koristit ćemo se funkcijom `glBegin(GL_TRIANGLE_FAN)` koja nam omogućava crtanje zajednički povezanih trokuta koji stvaraju mnogokut. S obzirom da će naš mnogokut imati puno vrhova biti će nalik kružnici. Pri crtanju svakog trokuta morat ćemo definirati koordinate trokuta što ćemo učiniti koristeći funkciju `glVertex2f()`. Varijabla `DEGTORAD` sadrži omjer između stupnjeva i radijana, varijabla `r` sadrži polumjer pojedine čestice, a varijablu `M2P` koristimo pri pretvorbi radijana u piksele, ta konverzija nam je potrebna jer u nastavku koristimo Box2D knjižnicu.

```
      void drawParticle(b2Vec2 center, float r, float angle, float red,
float green, float blue, float alpha){
          glColor4f(red,green,blue,alpha);

          glPushMatrix();
              glTranslatef(center.x*M2P,center.y*M2P,0);
              glRotatef(angle*180.0/M_PI,0,0,1);
              glBegin(GL_TRIANGLE_FAN);
                  for (float a = 0; a < 360 * DEGTORAD; a += 10 *
DEGTORAD)
                      glVertex2f( r*sinf(a)*M2P, r*cosf(a)*M2P );
              glEnd();
          glPopMatrix();
      }
```

Kod 2.4: Prikaz drawParticle() funkcije koji sadrži OpenGL funkcije za crtanje čestica.

2.4. Multimedijalna knjižnica SDL 2.0

SDL (eng. Simple DirectMedia Layer) je prilično korisna knjižnica pisana u C-u koja znatno olakšava interaktivni rad i općenito pisanje OpenGL aplikacija, pošto sam OpenGL ne daje podršku za rad s prozorima. Osim što pruža podršku za rad s prozorima ova knjižnica omogućava zadavanje koordinata putem miša u prozoru, pomicanje objekata interaktivno s mišem, upravljanje događajima i slično. Aplikacije stvorene s ovom knjižnicom moguće je pokretati na bilo kojem operacijskom sustavu. [11]

U ovom radu korištena je najnovija verzija SDL-a tj. SDL 2.0 s dodatnom službenom knjižnicom pod nazivom SDL_image koja pruža podršku za različite formate slika, a korištena je prilikom učitavanja tekstura.

Kod pod brojem 2.5 kopiran je iz main.cpp datoteke, a sadrži inicijalizaciju SDL 2.0 knjižnice i naredbe uz pomoć kojih se stvara prozor. Funkcijom `SDL_Init()` inicijaliziramo SDL knjižnicu, ova funkcija mora biti pozvana prije bilo koje druge SDL funkcije. Za filtriranje linearnih tekstura koristimo naredbu `SDL_SetHint()`. Stvaranje prozora započinjemo naredbom `SDL_CreateWindow()`. Ova funkcija prima parametre poput naziva prozora,

mjesta gdje ćemo smjestiti prozor, širinu i visinu prozora. `SDL_GL_CreateContext()` naredbu koristimo za stvaranje prozora unutar OpenGL-a. Posljednja naredba koda `SDL_Event` omogućava nam stvaranje strukture koja će biti zadužena za pratnju stanja događaja koji će se dešavati prilikom interakcije korisnika s našim programom.

```
SDL_Init(SDL_INIT_EVERYTHING);
if( !SDL_SetHint( SDL_HINT_RENDER_SCALE_QUALITY, "1" ) )
      {
        printf( "Warning: Linear texture filtering not enabled!" );
      }
SDL_Window * screen = SDL_CreateWindow("Particles",0,0,WIDTH,
HEIGHT,SDL_WINDOW_OPENGL);

SDL_GL_CreateContext( screen );
SDL_Event event;
```

Kod 2.5: Prikaz stvaranja prozora uz pomoć SDL 2.0 knjižnice.

Način na koji određujemo šta će se dešavati prilikom pritiska određene tipke na tipkovnici vidljiv je u kodu 2.6. Pritisak tipke 'Esc' na tipkovnici uzrokovati će promjenu bool varijable `running` u false, što će u konačnici značiti izlaskom iz programa. Naredbom `TwEventSDL` služimo se prilikom rada sa AntTweakBar GUI-jem te nam ta naredba omogućuje mijenjanje određenih vrijednosti unutar našeg GUI-a.

```
while(SDL_PollEvent(&event))
{
      handled = TwEventSDL(&event, SDL_MAJOR_VERSION,
SDL_MINOR_VERSION);
      if(!handled){
            switch(event.type)
            {
                  case SDL_QUIT:
                        running=false;
                        break;
                  case SDL_KEYDOWN:
                        switch(event.key.keysym.sym)
                        {
                              case SDLK_ESCAPE:
                              running=false;
                              break;
                        }
                        break;
            }
        }
}
```

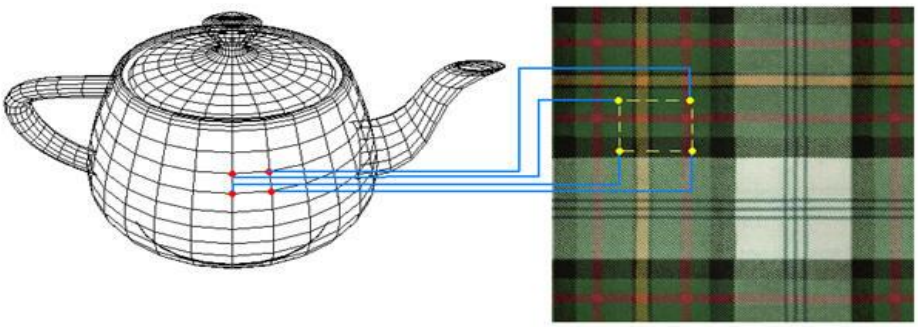
Kod 2.6: Upravljanje događajima uz pomoć SDL 2.0 knjižnice.

Teksture u računalnoj grafici značajno obogaćuju prikaze trodimenzionalnih scena i omogućavaju stvaranje realnije vizualne scene. U računalnoj grafici pod pojmom tekstura smatramo svaku dvodimenzionalnu sliku. U današnje vrijeme preslikavanje tekstura na poligone je sklopovski podržano te se iscrtavanje objekata s teksturama odvija vrlo brzo. [12]

Postupak teksturiranja (eng. texture mapping) podrazumijeva preslikavanje tekstura nad površinom virtualnog objekta. Preslikavanje (eng. mapping) obavljamo tako da se svakom pojedinom vrhu (eng. vertex) poligona dodijeli koordinata koja odgovara koordinatama pridruženog elementa teksture koji se nalazi u polju teksture. Preslikavanje možemo obavljati na različite načine, a koordinate mogu biti i dinamične tj. mogu se mijenjati ovisno o vremenu i pogledu.

Teksture najčešće koristimo kako bi određenom trodimenzionalnom obliku pridodali određenu boju u nekoj točki. No osim samog preslikavanja teksture nam omogućuju i predstavljanje različitih obilježja tijela koje želimo prikazati. Tako na primjer tekstura nam može pomoći u određivanju sjajnosti tijela, hrapavosti tijela, pa čak i dobiti izgled neravnina na ravnoj površini. [13]

Teksturiranje koje će biti korišteno u ovom primjeru je Osnovno teksturiranje. To je najjednostavnija vrsta teksturiranja u kojoj se 2D slika preslikava na 2D površinu.



Slika 2.1: Prikaz preslikavanja teksture. [13]

U našem primjeru koristit ćemo knjižnicu SDL_image 2.0 za učitavanje PNG (eng. Portable Network Graphics) format slike koji ćemo kasnije pretvoriti u OpenGL texturu. SDL_image 2.0 je knjižnica namijenjena isključivo za učitavanje slika. Slike učitava kao SDL površine (eng. surface) i teksture, a podržava sljedeće formate: BMP, GIF, JPEG, LBM, PCX, PNG, PNM, TGA, TIFF, WEBP, XCF, XPM, XV. [14]

Na kodu 2.7 moguće je vidjeti implementaciju funkcije `LoadTexture()` koja se koristi za učitavanje slike i stvaranje teksture u OpenGL-u uz pomoć SDL_image knjižnice. Uz pomoć SDL_image funkcije `IMG_Load()` učitavamo PNG format slike, te je nakon toga predajemo SDL površini (eng. surface). Sada kada imamo učitanu sliku treba pregledati kakav

je format boja slike, odnosno da li slika koristi običan RGB (24 bit-a) ili RGBA (32 bit-a) format boja. Sljedeći korak koji trebamo napraviti je generiranje imena za našu teksturu putem naredbe `glGenTexture()`. Nakon što imamo ime teksture moramo OpenGL-u dati naredbu za korištenje navedenog imena teksture (`glBindTexture()`). Kako bi prijašnji `SDL_Surface` pretvorili u OpenGL teksturu koristimo funkciju `glTexImage2D()` nakon čega dobivamo željenu teksturu u OpenGL-u. Funkcija `glTexParametri()` služi isključivo za određivanje načina na koji će tekstura biti prikazana.

```
GLuint LoadTexture(char const *filename,int *textw,int *texth) {
        SDL_Surface *surface;
        GLuint textureid;
        int mode;

        surface = IMG_Load(filename);

        if (!surface) {
                return 0;
        }

        if (surface->format->BytesPerPixel == 3) {
                mode = GL_RGB;
        } else if (surface->format->BytesPerPixel == 4) {
                mode = GL_RGBA;
        } else {
                SDL_FreeSurface(surface);
                return 0;
        }

        *textw=surface->w;
        *texth=surface->h;

        glGenTextures(1, &textureid);
        glBindTexture(GL_TEXTURE_2D, textureid);
        glTexImage2D(GL_TEXTURE_2D, 0, mode, surface->w, surface-
>h, 0, mode, GL_UNSIGNED_BYTE, surface->pixels);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

        SDL_FreeSurface(surface);

        return textureid;

}
```

Kod 2.7: Dio koda koji sadrži inicijalizacija i dodavanje nekih varijabli unutar AntTweakBar GUI-a.

2.5. Fizička knjižnica Box2D

Box2D je besplatna dvodimenzionalna knjižnica otvorenog koda, a služi za simulaciju fizike. Napisao ju je Erin Catto za konferenciju programera 2006. godine, a pisana je u programskom jeziku C++. U zadnjih se par godina znatno razvijala i optimizirala te je danas postala jako popularna kod programiranja manjih dvodimenzionalnih programa ili igara. Popularne igre programirane ovom knjižnicom su 'Angry Birds' i 'Limbo'. Ova knjižnica pruža jednostavnost i brzinu pri programiranju. Bazira se isključivo na fiziku i ne poznaje nikakve naredbe iz svijeta grafike što omogućava korisnicima da se baziraju isključivo na fiziku koju kasnije samo objedinjuju s OpenGL-om. [15]

U ovom projektu korištena je trenutno najnovija verzija Box2D knjižnice, a to je verzija 2.3.0. Knjižnica je korištena za stvaranje realne okoline u kojoj se nalaze čestice, odnosno okoline u kojoj djeluje gravitacijska sila koja je jednaka Zemljinoj, za dodjeljivanje određene mase svakoj čestici te za omogućavanje djelovanja sila poput vjetra na sustav čestica.

Za inicijalizaciju Box2D-a potrebno je stvoriti svijet koji će sadržavati sve naše objekte, odnosno sve naše sustave čestica koji će imati određena fizička svojstva. Kod pod brojem 2.2, kojega smo još ranije spomenuli sadrži naredbu (posljednja naredba u kodu) za stvaranje svijeta (eng. world) s gravitacijom od 9.81 $\frac{m}{s^{-2}}$ unutar kojeg će se nalaziti sve naše čestice. Ostale funkcije i naredbe same knjižnice biti će objašnjene u nastavku prilikom objašnjavanja programskog dijela završnog rada. [16]

2.6. AntTweakBar GUI knjižnica

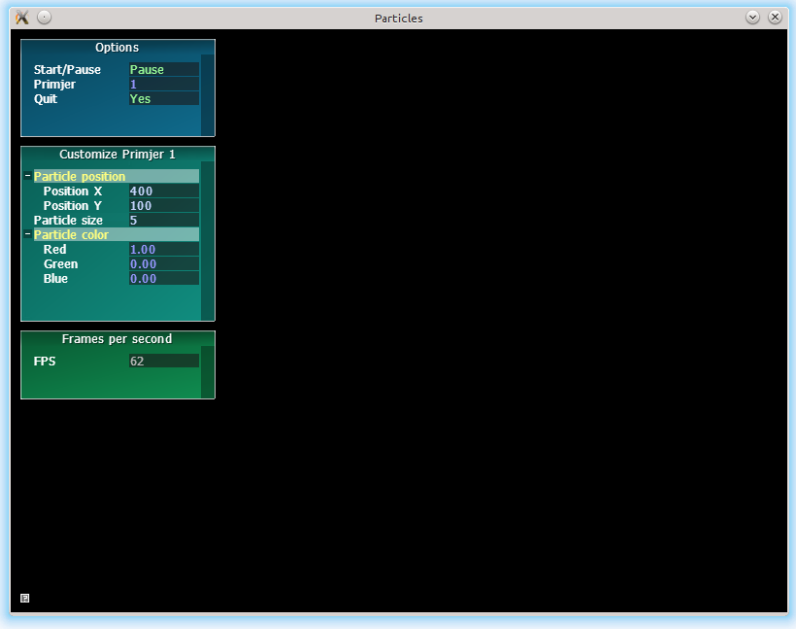
AntTweakBar je malena i jednostavna knjižnica pisana u C++ koja programerima pruža mogućnost brze i jednostavne implementacije GUI-a unutar njihovih grafičkih aplikacija koje se baziraju na OpenGL-u ili DirectX-u. C++ varijable su većinom smještene unutar GUI-a što omogućava korisniku jednostavnu interakciju među tim varijablama. Varijable se prikazuju unutar aplikacije uz pomoć prozora koji se naziva 'tweak bar'. Ova knjižnica je nastala s ciljem da omogući jednostavan i brz način mijenjanja parametara unutar grafičke aplikacije te za vizualizaciju scene koje nastaje prilikom izmjene parametara u realnom vremenu. [17]

Pri izradi ovog projekta korištena je izmijenjena verzija najnovije verzije 1.16 AntTweakBar knjižnice jer knjižnica trenutno ne podržava podršku za SDL 2.0. Na prikazanom kodu 2.8 vidimo da je inicijalizirati ovu knjižnicu zbilja jednostavno i lagano. Inicijalizacija počinje `TwInit()` funkcijom kojoj govorimo da je naša aplikacija programirana u OpenGL-

u. Zatim funkciji `TwWindowSize()` šaljemo podatke o veličine našeg prozora. `TwNewBar()` stvara novi bar pod nazivom 'Options' unutar kojega će biti smještene sve naše varijable. `TwDefine()` je funkcija u kojoj dodajemo poruku koju želimo da bude prikazana kada korisnik klikne na 'help' gumb. Varijablu dodajemo putem funkcije `TwAddVarRW()`, a zadnja dva slova 'RW' označavaju da varijablu možemo pročitati (eng. read) i promijeniti odnosno zapisati (eng. write).

```
TwInit(TW_OPENGL, NULL);

    // Tell the window size to AntTweakBar
TwWindowSize(WIDTH, HEIGHT);

    // Create a tweak bar
bar = TwNewBar("Options");

TwDefine("    GLOBAL    help='Particle    system    with    physics
simulation.\nPress [Esc] to quit the program.\n' "); // Message added to
the help bar.

TwAddVarRW(bar,     "Pause",     TW_TYPE_BOOL32,     &pause,     "
label='Start/Pause ' true='Start' false='Pause' help='Start/Pause' ");

TwAddVarRW(bar, "Primjer", TW_TYPE_INT32, &primjer, " min=1 max=10
step=1 label='Primjer' ");

TwAddVarRW(bar, "Quit", TW_TYPE_BOOL32, &running, " label='Quit '
true='Yes' false='No' key='ESC' help='Exit' ");
```

Kod 2.8: Dio koda koji sadrži inicijalizacija i dodavanje nekih varijabli unutar AntTweakBar GUI-a.



Slika 2.1: Prikaz GUI-a koji nastaje korištenjem AntTweakBar-om.

3. Sustav čestica

Sustav čestica je skup puno sićušnih čestica koje zajedno predstavljaju neki neizraziti objekt. Tijekom određenog vremenskog razdoblja događa se generiranje čestica unutar sustava, nakon čega nastaju pokreti i promjene pojedinih čestica sve dok na posljetku čestice ne umru unutar sustava.

Od ranih 1980-tih čestični sustavi su korišteni u brojnim video igrama, u stvaranju animacija, digitalnih umjetničkih djela te u raznim sustavima za modeliranje različitih nepravilnih vrsta prirodnih pojava, kao što su oblaci, vatra, vodopadi, magla, trava, mjehurići i slično.

Ovo će se poglavlje baviti implementacijom strategija za izgradnju čestičnog modela. Počet ćemo s jednostavnim primjerom gdje se nalazi samo jedna čestica bez teksture, a završiti s sustavom čestica na koji djeluje tekstura. [18]

3.1. Zašto su nam potrebni čestični sustavi

Definirali smo sustav čestica kao skup nezavisnih objekata, koji su često zastupljeni jednostavnim oblicima poput točke ili forme. Zašto je to važno? Dakako za mogućnost modeliranja nekih od navedenih fenomena poput eksplozije, što je atraktivno i potencijalno korisno. Ali zapravo postoji još jedan bolji razlog zašto učiti i koristiti čestične sustave. Ako želimo biti bilo gdje unutar života koda sustava morat ćemo raditi sa sustavima mnogih stvari. Odnosno željeti ćemo gledati loptice koje skakuću, ptice kako lete, ekosustave koji evoluiraju i svakakve stvari koje se dešavaju u množini. [18]

Na samom početku početi ćemo se baviti fleksibilnim količinama elemenata. Ponekad ćemo imati jednu česticu ponekad ćemo imati deset čestica, a ponekad ćemo imati deset tisuća čestica. Umjesto pisanja strukture za svaku pojedinu česticu napisat ćemo klasu koja će moći opisivati cijeli sustav čestica s ciljem da naš program bude pregledniji i efikasniji.

Sljedeće će poglavlje opisivati implementaciju naše čestice i stvaranje jedne jednostavne čestice.

3.2. Stvaranje jedne čestice (primjer 1)

U primjeru 1 izraditi ćemo jednostavnu implementaciju koja će sadržavati i baviti se samo s jednom česticom. Kako bi nam kasnije bilo jasnije kako sustavi čestica rade.

Na samom početku stvoriti ćemo klasu koja će opisivati našu česticu. Kao što se može vidjeti u kodu 3.1 naša `Particle()` klasa će sadržavati tijelo čestice (`p_body`) koje mora biti definirano za svako tijelo na koje namjeravamo koristiti Box2D fizičku knjižnicu, veličinu čestice (`p_size`), koordinate točke lokacije (`p_pos.x i p_pos.y`), vrijeme trajanja čestice (`p_life`), svjetlinu čestice (`p_alpha`), kontakt koji se dešava kada se dvije čestice dotaknu (`p_lastContact`) te broj (`p_num`) s kojim ćemo moći identificirati vrstu naše čestice. Ove zadnje dvije varijable će nam biti korisne kod dodirivanja čestica i bit će objašnjene naknadno u primjeru 6.

U navedenom kodu moguće je i vidjeti metode koje će biti korištene za stvaranje čestice

```
class Particle{
    public:
        b2Body* p_body;
        b2Vec2 p_pos;
        int p_radius;
        Uint32 p_life;
        float p_alpha;
        b2Contact* p_lastContact;
        int p_num;

    public:
        Particle();
        void createParticle(Uint32 life, int pos_x, int pos_y, int
radius);
        void drawParticle(b2Vec2 center, float r, float angle);
        void updateParticle(Uint32 decLife, float wind_force,
float wind_position);
        void updateParticleNormal(Uint32 decLife);
        void deleteParticle();
        bool isDead();
};
```

Kod 3.1: Dio koda koji sadrži definiciju klase Particle, a nalazi se u direktoriju external/particle.h.

(**createParticle**), crtanje čestice (**drawParticle**), ažuriranje čestice (**updateParticle**), za brisanje čestice (**deleteParticle**) te za provjeru je li naša čestica mrtva (**isDead**).

Na sljedećem kodu (3.2) nalazi se konstruktor klase `Particle` koji inicijalizira osnovne varijable čestice.

```
Particle::Particle(){
    p_body = NULL;
    p_radius = 5;
    p_pos.x = 400;
    p_pos.y = 100;
    p_alpha = 1.0;
    p_life = 0.0;
    p_lastContact = NULL;
    p_num = 0;
}
```

Kod 3.2: Dio koda koji sadrži konstruktor klase Particle, a nalazi se u direktoriju external/particle.cpp.

Na kodu broj 3.3 moguće je vidjeti metodu `createParticle` koja nam služi pri izradi čestice. U ovom kodu susrećemo se s mnogo različitih Box2D metoda i objekata. Da bi stvorili tijelo potrebno je prije toga stvoriti objekt koji će definirati to tijelo (`myBodyDef`). Nakon toga pridodajemo vrstu i poziciju definiciji objekta tijela. Kad smo definirali objekt tijela tada možemo stvoriti tijelo uz pomoć metode `CreateBody()`. U našem primjeru je vrsta tijela definirana kao `b2_dynamicBody` što znači da će se tijelo kretati i sudarati sa svim ostalim tijelima, a da će masa tijela biti različita od nule. Kako bi tijelo stvorili na točnoj lokaciji potrebno je x i y koordinate pomnožiti sa `P2M` varijablom zbog konverzije piksela u radijane. S obzirom da stvaramo čestice koristimo oblik `circleShape` koji nam omogućuje stvaranje 'lopte' te joj dodjeljujemo željenu veličinu naredbom `circleShape.m_radius`.

```
    void Particle::createParticle(Uint32 life, int pos_x, int pos_y,
int radius){
        p_life = life;
        p_alpha = 1.0;
        p_pos.x = pos_x;
        p_pos.y = pos_y;
        p_radius = radius;

        b2BodyDef myBodyDef;
        myBodyDef.type = b2_dynamicBody;
        myBodyDef.position.Set(p_pos.x*P2M,p_pos.y*P2M);
        p_body = getWorld()->CreateBody(&myBodyDef);

        b2CircleShape circleShape;
        circleShape.m_p.Set(0, 0);
        circleShape.m_radius = p_radius*P2M;

        b2FixtureDef myFixtureDef;
        myFixtureDef.shape = &circleShape;
        myFixtureDef.density = 0.1;
        myFixtureDef.friction = 0.0;
        myFixtureDef.restitution = 0.0;
        p_body->CreateFixture(&myFixtureDef);
        p_body->SetUserData(&p_num);
    }
```

Kod 3.3: Dio koda koji sadrži opis metode createParticle klase Particle, a nalazi se u direktoriju external/particle.cpp.

Objekt `myFixtureDef` omogućuje nam pričvršćivanje oblika za tijelo. Ovom naredbom našoj čestici dodjeljujemo gustoću (`density`), trenje (`friction`) i sposobnost odbijanja (`restitution`). Kako bismo kasnije mogli odrediti koji se objekti sudaraju koristimo naredbu `SetUserData()`, više o tome u primjeru 6.

```
      void drawParticle(b2Vec2 center, float r, float angle, float red,
float green, float blue, float alpha){
          glColor4f(red,green,blue,alpha);

          glPushMatrix();
              glTranslatef(center.x*M2P,center.y*M2P,0);
              glRotatef(angle*180.0/M_PI,0,0,1);
              glBegin(GL_TRIANGLE_FAN);
                  for (float a = 0; a < 360 * DEGTORAD; a += 10 *
DEGTORAD)
                      glVertex2f( r*sinf(a)*M2P, r*cosf(a)*M2P );
              glEnd();
          glPopMatrix();
      }
```

Kod 3.4: Dio koda koji sadrži opis metode drawParticle klase Particle, a nalazi se u direktoriju external/utils.cpp.

Cijeli ovaj kod pod brojem 3.4 objašnjen je u poglavlju gdje se opisuje OpenGL, te se ovdje neće dodatno opisivati.

```
      void Particle::updateParticle (Uint32 decLife){

          p_body->ApplyForce( b2Vec2(randFrom(-1.0, 1.0),randFrom(-1.0,
1.0)), p_body->GetWorldCenter() );

          p_life -= decLife;
          if(p_life<=0.0)
              getWorld()->DestroyBody(p_body);

      }
```

Kod 3.5: Dio koda koji sadrži opis metode updateParticleNormal klase Particle, a nalazi se u direktoriju external/particle.cpp.

Čestice ažuriramo uz pomoć `updateParticle()` metode. Ona nam omogućava dodavanje pojedine sile na česticu, smanjenje života čestice i brisanje čestice ukoliko je čestica mrtva. Funkciju `randFrom(-1.0, 1.0)` koristimo za generiranje slučajnih brojeva između -1.0 i 1.0. Njezin kod vidljiv je u kodu 3.6

```
double randFrom(double min, double max)
{
    double range = (max - min);
    double div = RAND_MAX / range;
    return min + (rand() / div);
}
```

Kod 3.6: Dio koda koji sadrži opis funkcije randFrom, a nalazi se u direktoriju external/utils.cpp.

Sljedeći kod 3.7 prikazuje posljednje dvije metode klase Particle koje će se koristiti pri stvaranju čestice.

```
void Particle::deleteParticle(){
    getWorld()->DestroyBody(p_body);
}
bool Particle::isDead(){
    if(p_life > 0.0){
        return false;
    }else{
        return true;
    }
}
```

Kod 3.7: Dio koda koji sadrži opis metoda deleteParticle i isDead klase Particle, a nalazi se u direktoriju external/particle.cpp.

```
if(TwGetBarByName("Customize Primjer 1") == NULL){
      TWbar1();

}
if(pause!=false){

      if(SDL_GetTicks()-old>=1.0){

            if(p[0].isDead()==false){

                  p[0].updateParticleNormal(1);
            }else{

                  p[0].createParticle(life,posX,posY,size);
            }
            old=SDL_GetTicks();
      }
}
```

Kod 3.8: Dio koda koji sadrži naredbe za stvaranje jedne čestice, nalazi se unutar main.cpp-a.

Kod pod brojem 3.8 sadrži metode za stvaranje GUI-a i jedne čestice. Naredbom `if(TwGetBarByName("Customize Primjer 1") == NULL)` pregledavamo je li naš TwBar već stvoren, a ukoliko nije stvaramo novi. Naredbom `if(pause!=false)` pregledavamo da li je stisnuta tipka pause, ukoliko je program će mirovati. Nadalje okidamo

timer nakon svake milisekunde koja je prošla `if(SDL_GetTicks()-old>=1.0)` te pregledavamo je li naša čestica živa. Ukoliko je živa ažuriramo česticu (smanjujemo joj životni vijek i dodajemo joj silu kako bi se kretala u različitim smjerovima), a ako je mrtva stvaramo novu česticu.

```
      void TWbar1(){
          primjer_1 = TwNewBar("Customize Primjer 1");
          TwDefine(" 'Customize Primjer 1' position='10 120'");
          TwDefine(" 'Customize Primjer 1' size='200 180'");

          TwAddVarRW(primjer_1, "PosX", TW_TYPE_INT32, &posX, " min=1
max=800 step=1 group='Particle position' label='Position X' ");
          TwAddVarRW(primjer_1, "PosY", TW_TYPE_INT32, &posY, " min=1
max=600 step=1 group='Particle position' label='Position Y' ");

          TwAddVarRW(primjer_1, "Size", TW_TYPE_FLOAT, &size, " min=1.0
max=10.0 step=0.5 label='Particle size' ");


          TwAddVarRW(primjer_1, "Red", TW_TYPE_FLOAT, &red, " min=0.0
max=1.0 step=0.01 group='Particle color' label='Red' ");
          TwAddVarRW(primjer_1, "Green", TW_TYPE_FLOAT, &green, "
min=0.0 max=1.0 step=0.01 group='Particle color' label='Green' ");
          TwAddVarRW(primjer_1, "Blue", TW_TYPE_FLOAT, &blue, " min=0.0
max=1.0 step=0.01 group='Particle color' label='Blue' ");


          fpsBar = TwNewBar("Frames per second");
          TwDefine(" 'Frames per second' position='10 310'");
          TwDefine(" 'Frames per second' size='200 20'");
          TwAddVarRO(fpsBar, "FPS", TW_TYPE_INT32, &fps, " label='FPS'
");
      }
```

Kod 3.9: Dio koda koji sadrži funkciju TWbar1, a služi za stvaranje GUI za primjer 1, nalazi se unutar direktorija external/gui.cpp.

```
      if(tmp->GetFixtureList()->GetShape()->GetType()==0){

         b2CircleShape* c=((b2CircleShape*)tmp->GetFixtureList()-
>GetShape());
         drawParticle(tmp->GetWorldCenter(), c->m_radius, tmp-
>GetAngle(), red, green, blue, alpha);

      }
```

Kod 3.10: Dio koda koji sadrži naredbe za stvaranje jedne čestice, nalazi se unutar display() funkcije unutar main.cpp-a.

Jednom kada smo sve to uspjeli napraviti preostaje nam još samo da unutar display() funkcije prikažemo našu česticu. Pošto je naša čestice krug koristit ćemo naredbu `if(tmp->GetFixtureList()->GetShape()->GetType()==0)` koja nam omogućuje

hvatanje svih krugova unutar Box2D svijeta, a nakon toga metodom `drawParticle()` crtamo našu česticu.

Slika 3.1: Prikaz stvaranja jedne čestice (primjer 1) iz programa.

3.3. Stvaranje sustava čestica (primjer 2)

Za stvaranje sustava čestica koristit ćemo se poljem (eng. array) u koji ćemo spremati sve naše čestice (`Particle p[MAX]`). Odabrali smo `MAX=100` zbog toga jer će svaka naša čestica trajati 100 ms te će nam biti potrebno da imamo 100 čestica kako bi mogli svake milisekunde stvarati novu česticu. Problem koji će se dogoditi je taj da ćemo morati znati mjesto u polju na kojem se ne iskorištena čestica nalazi kako bi mogli stvoriti novu česticu. Dakle trebat će nam dodatna funkcija `unusedParticle()` i varijabla `LastUsedParticle` kako bi mogli pronaći broj ne iskorištene čestice.

```
int LastUsedParticle = 0;
const int MAX = 200;
Particle p[MAX];
int unusedParticle(){
    for(int i=LastUsedParticle; i<MAX; i++){
        if (p[i].isDead()){
            LastUsedParticle = i;
            return i;
        }
    }
    for(int i=0; i<LastUsedParticle; i++){
        if (p[i].isDead()){
            LastUsedParticle = i;
            return i;
        }
    }
    return 0;
}
```

Kod 3.11: Dio dodatnog koda kojeg ćemo koristit u primjeru 2, a nalazi se unutar main.cpp-a.

```
if(TwGetBarByName("Customize Primjer 1") != NULL){
      TWbar1Delete();
}
if(TwGetBarByName("Customize Primjer 2") == NULL){
      TWbar2();
}

if(pause!=false){

      if(SDL_GetTicks()-old>=1.0){

        LastUsedParticle = unusedParticle();
        p[LastUsedParticle].createParticle(life,posX,posY,size);

        for(int i=0; i<MAX; i++){
            if(p[i].isDead()==false){
                  p[i].updateParticle(1);
            }
        }
        old=SDL_GetTicks();
     }
}
```

Kod 3.12: Dio koda koji sadrži naredbe za stvaranje sustava čestica, nalazi se unutar main.cpp-a.

Ostatak koda sličan je kao i u prvom primjeru samo se ovdje koristimo prijašnje objašnjenom funkcijom za dobivanje broja ne iskorištene čestice. Ovdje će biti potrebno pregledati da li postoji `TWbar1` te ukoliko postoji izbrisati ga i stvoriti novi jer će se koristiti različiti naziv u postavkama u odnosu na `TWbar1`. Kako je razlika između `TWbar1` i `TWbar2` jako malena kod nećemo dodatno objašnjavati.

```
void TWbar1Delete(){
    TwDeleteBar(primjer_1);
    TwDeleteBar(fpsBar);
}
```

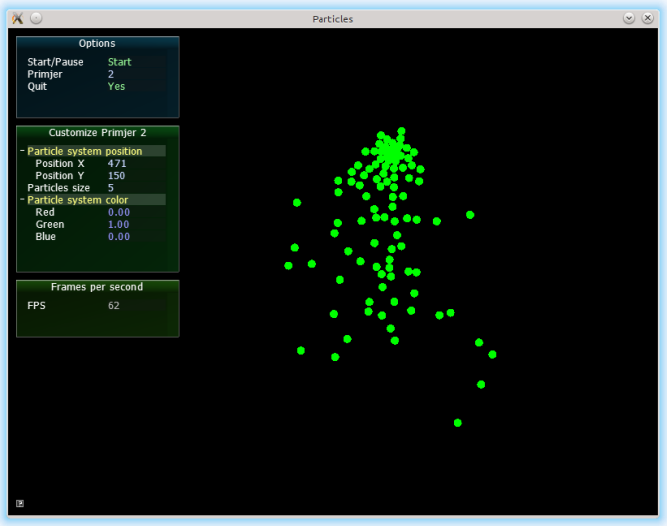
Kod 3.13: Dio koda koji sadrži naredbe za brisanje TWbar1, nalazi se u direktoriju external/utils.cpp.

3.4. Stvaranje sustava čestica pritiskom miša (primjer 3)

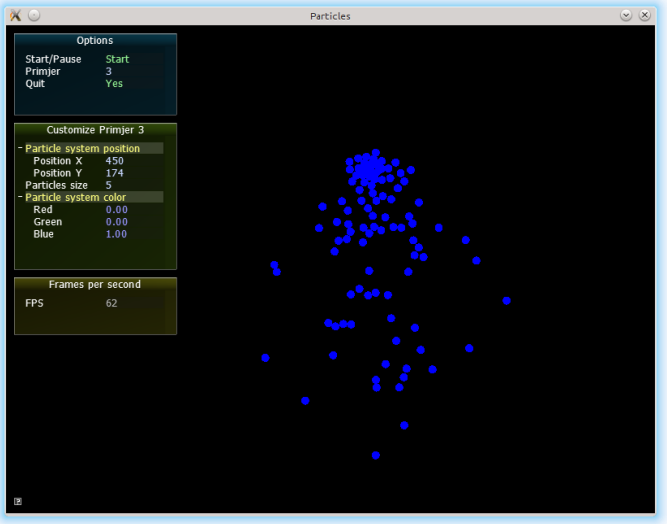
Pri stvaranju sustava čestica pritiskom na miš koristimo se istom implementacijom kao i u primjeru 2 osim što ćemo se ovdje koristiti pomoćnom varijablom `crtaj` koja će nam služiti kao okidač kad budemo stisnuli tipku miša. Za kontrolu pritiska miša koristimo se SDL naredbom `event.button.button == SDL_BUTTON_LEFT` što je sve vidljivo na kodu 3.14.

```
	if(pause!=false){

		if(SDL_GetTicks()-old>=1.0){

			if(crtaj == true){
				LastUsedParticle = unusedParticle();

				p[LastUsedParticle].createParticle(life,posX,pos
				Y,size);
			}
			for(int i=0; i<MAX; i++){
				if(p[i].isDead()==false){
					p[i].updateParticle(1);
				}
			}
			old=SDL_GetTicks();
		 }
	}
	while(SDL_PollEvent(&event)){

		handled = TwEventSDL(&event, SDL_MAJOR_VERSION,
	SDL_MINOR_VERSION);

		if(!handled){

			switch(event.type)
			{
				case SDL_QUIT:
					running=false;
				break;
				case SDL_MOUSEBUTTONDOWN:
					if(event.button.button ==
SDL_BUTTON_LEFT){
						posX = event.button.x;
						posY = event.button.y;

						crtaj = true;
					}
				break;
				case SDL_KEYDOWN:

					switch(event.key.keysym.sym)
					{
						case SDLK_ESCAPE:
							running=false;
						break;
					}
				break;
			 }
		 }
	}
```

Kod 3.14: Dio koda koji sadrži naredbe za stvaranje sustava čestica pritiskom miša, nalazi se unutar main.cpp-a.



Slika 3.2: Prikaz stvaranja jedne čestice (primjer 2) iz programa.



Slika 3.3: Prikaz stvaranja jedne čestice (primjer 3) iz programa.

3.5. Stvaranje više sustava čestica pritiskom miša (primjer 4)

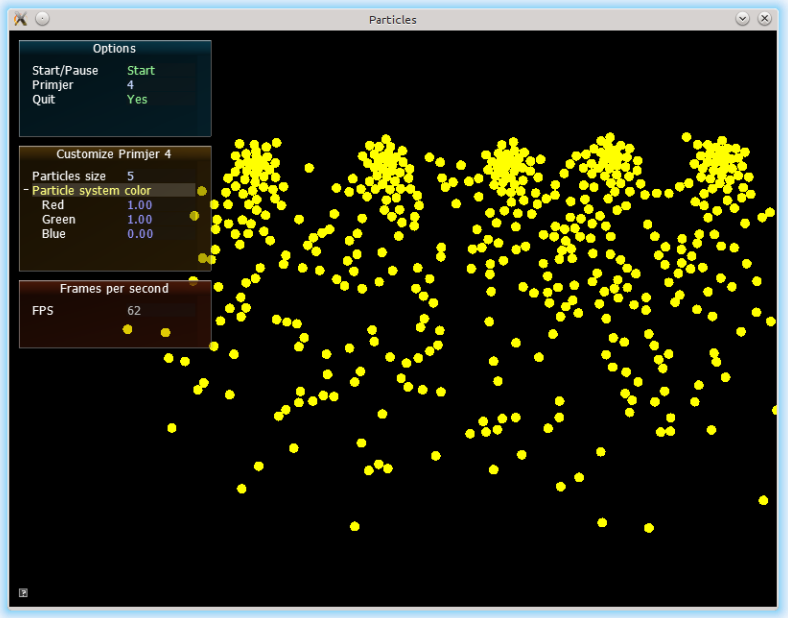
Kod ovog primjera biti će potrebna nova implementacija objekta pa ćemo tu koristiti dvodimenzionalnu matricu (eng. 2d array) `Particle Systems[Smax][MAX]`. Iz koda 3.15 moguće je vidjeti da je najveći broj sustava pet pošto je `Smax = 5`. Slično kao u primjeru 3 koristimo pomoćnu bool varijablu `crtaj2` koja nam služi kao okidač pri pritiskom tipke miša. Varijablu `tmp2` ćemo povećavati svaki put prilikom pritiska tipke miša. Koristi ćemo je kod `display()` funkcije kako bi saznali koliko sustava čestica moramo nacrtati. U sljedeća tri koda moguće je vidjeti potpuni kod primjera 4.

```
bool crtaj2 = false;
int tmp2 = 0;
const int MAX = 200;
const int Smax = 5;
int LastUsedParticleSystems[Smax];
Particle Systems[Smax][MAX];
```

Kod 3.15: Dio koda koji sadrži inicijalizaciju koda kod stvaranja više sustava čestica pritiskom miša, nalazi se unutar main.cpp-a.

```
if(tmp->GetFixtureList()->GetShape()->GetType()==0){
                                b2CircleShape* c=((b2CircleShape*)tmp-
>GetFixtureList()->GetShape());
                                drawParticle(tmp->GetWorldCenter(), c-
>m_radius, tmp->GetAngle(), red, green, blue, alpha);
                            }
```

Kod 3.16: Dio koda koji sadrži kod unutar display() funkcije koji se koristi za stvaranja više sustava čestica pritiskom miša, nalazi se unutar main.cpp-a.



Slika 3.4: Prikaz stvaranja više sustava čestica mišem (primjer 4) iz programa.

```
	if(pause!=false){
		if(SDL_GetTicks()-old>=1.0){

		if(crtaj2 == true){
			for(int i=0; i<tmp2; i++){
			LastUsedParticleSystems[i] = unusedParticleSystem(i);
			Size[i] = size;
			Systems[i][LastUsedParticleSystems[i]].createParticle(
			life,PosX[i],PosY[i],Size[i]);

			}
			for(int j=0; j<Smax; j++){
				for(int i=0; i<MAX; i++){

					if(Systems[j][i].isDead()==false){
					Systems[j][i].updateParticle(1);
					}
				}
			}
		}
		old=SDL_GetTicks();
		}
	}
	while(SDL_PollEvent(&event)){

		handled = TwEventSDL(&event, SDL_MAJOR_VERSION,
	SDL_MINOR_VERSION);

		if(!handled){

			switch(event.type)
			{
				case SDL_QUIT:
					running=false;
				break;
				case SDL_MOUSEBUTTONDOWN:
					if(event.button.button ==
SDL_BUTTON_LEFT){
						PosX[tmp2] = event.button.x;
						PosY[tmp2] = event.button.y;
						Size[tmp2] = 5;
						tmp2++;
						if(tmp2>5)tmp2=0;
						crtaj2 = true;

					}
				break;
				case SDL_KEYDOWN:
					switch(event.key.keysym.sym)
					{
						case SDLK_ESCAPE:
							running=false;
						break;
					}
				break;
				}
			}
		}
```

Kod 3.17: Dio koda koji sadrži naredbe za stvaranje više sustava čestica pritiskom miša, a nalazi se unutar main.cpp-a.

3.6. Stvaranje dva različita sustava čestica (primjer 5)

Za stvaranje nove vrste čestica korišten je objektno orijentirani princip nasljeđivanja (eng. inheritence). Nasljeđivanje nam pomaže kada već imamo definiciju nekog objekta, a potrebna nam je definicija sličnog objekta. Tada je moguće u definiciji novog objekta naslijediti već definirani prijašnji objekt, čime štedimo vrijeme za programiranje i prostor na disku.

Na kodu broj 3.18 moguće je vidjeti kako klasa `Box` nasljeđuje klasu `Particle` i kako polimorfizmom preopterećujemo metodu `createParticle()`. Polimorfizam je jedan od objektno orijentiranih principa koji nam omogućuje preopterećivanje metoda. Moguće je definirati nekoliko metoda istoga imena, a svaka će kao parametre primati objekte različitih tipova.

```
class Box : public Particle{
    public:
        int p_a;
    public:
        Box();
        void createParticle(Uint32 life, int pos_x, int pos_y, int
a, b2BodyType type);
};
```

Kod 3.18: Dio koda koji sadrži implementaciju Box klase, a nalazi se u direktoriju external/particle.h.

Definicije metode `createParticle()` i nova funkcija `drawBox()` su jako slične prijašnjim definicijama. Jedina razlika je u tome da će nam ova metoda i funkcija omogućiti crtanje kvadratića za razliku od prijašnjih koje su nam omogućavale crtanje krugova (čestica).

```
void drawBox(b2Vec2* points,b2Vec2 center, float angle, float red,
float green, float blue, float alpha){
    glColor4f(0.25,0.55,0.0,alpha);
    glPushMatrix();
        glTranslatef(center.x*M2P,center.y*M2P,0);
        glRotatef(angle*180.0/M_PI,0,0,1);
        glBegin(GL_QUADS);
            for(int i=0;i<4;i++)
                glVertex2f(points[i].x*M2P,points[i].y*M2P);
        glEnd();
    glPopMatrix();
}
```

Kod 3.19: Dio koda koji sadrži implementaciju metode drawBox(), a nalazi se u direktoriju external/utils.h.

```
      void Box::createParticle(Uint32 life, int pos_x, int pos_y, int a,
b2BodyType type){
          p_life = life;
          p_alpha = 1.0;
          p_pos.x = pos_x;
          p_pos.y = pos_y;
          p_a = a;
          p_num = 1;

          b2BodyDef myBodyDef;
          myBodyDef.type = type;
          myBodyDef.position.Set(p_pos.x*P2M,p_pos.y*P2M);
          p_body = getWorld()->CreateBody(&myBodyDef);

          b2PolygonShape shape;
          shape.SetAsBox(P2M*p_a,P2M*p_a);

          b2FixtureDef myFixtureDef;
          myFixtureDef.shape = &shape;
          myFixtureDef.density = 0.1;
          myFixtureDef.friction = 0.0;
          myFixtureDef.restitution = 0.0;
          p_body->CreateFixture(&myFixtureDef);
          p_body->SetUserData(&p_num);
      }
```

Kod 3.20: Dio koda koji sadrži implementaciju metode createParticle(), a nalazi se u direktoriju external/particle.h.

Nakon što smo definirali našu novu klasu potrebno je u glavnom programu stvoriti novi objekt te nacrtati navedena dva sustava čestica.

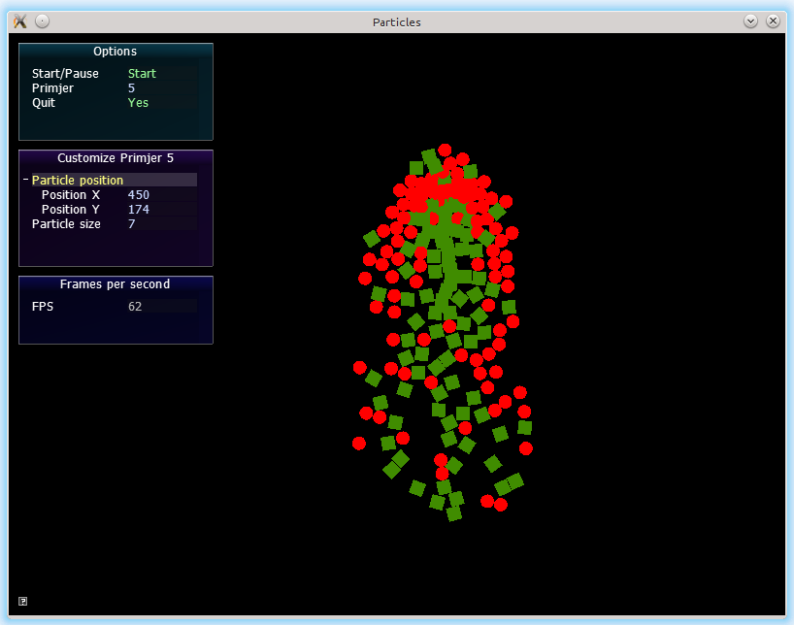
```
      if(pause!=false){
            if(SDL_GetTicks()-old>=1.0){
                  LastUsedParticle = unusedParticle();
                  p[LastUsedParticle].createParticle(life,posX,posY,radi
                  us);
                  for(int i=0; i<MAX; i++){
                        if(p[i].isDead()==false){
                              p[i].updateParticle(1);
                        }
                  }
                  LastUsedParticleBox = unusedParticleBox();
                  box[LastUsedParticleBox].createParticle(life,posX,posY
                  ,radius,b2_dynamicBody);

                  for(int i=0; i<MAX; i++){
                        if(box[i].isDead()==false){
                              box[i].updateParticle(1);
                        }
                  }
                  old=SDL_GetTicks();
            }
       }
```

Kod 3.21: Dio koda koji sadrži naredbe za stvaranje dva različita sustava čestica, a nalazi se unutar main.cpp-a.

Pošto se crtanje kvadratića ne obavlja na isti način kao i crtanje krugova potrebno je promijeniti i kod u display() funkciji.

```
      if(tmp->GetFixtureList()->GetShape()->GetType()==0){
                              b2CircleShape* c=((b2CircleShape*)tmp-
>GetFixtureList()->GetShape());
                              int a= *((int*)tmp->GetUserData());
                              if(a==2){
                                  drawCircle(tmp->GetWorldCenter(), c-
>m_radius, tmp->GetAngle(), red, green, blue, alpha);
                              }else{
                                  drawParticle(tmp->GetWorldCenter(), c-
>m_radius, tmp->GetAngle(), red, green, blue, alpha);
                              }
                          }else{
                              for(int i=0; i<4; i++){
                                  points[i] = ((b2PolygonShape*)tmp-
>GetFixtureList()->GetShape())->GetVertex(i);
                              }
                              drawBox(points, tmp->GetWorldCenter(),
tmp->GetAngle(), red, green, blue, alpha);
                          }
```

Kod 3.22: Dio koda koji sadrži kod unutar display() funkcije koji se koristi za stvaranja dva različita sustava čestica, a nalazi se unutar main.cpp-a.



Slika 3.5: Prikaz stvaranja dva različita sustava čestica (primjer pet) iz programa.

3.7. Odbijanje sustava čestica u odnosu na drugo tijelo (primjer 6)

Uobičajeno je misliti da se tijela sudaraju jedna s drugima, ali u Box2D-u su zapravo učvršćenja ta koja se koriste za otkrivanje kada dolazi do sudara. Sudari između dva tijela se mogu dogoditi na više različitih načina te upravo zbog toga Box2D sadrži mnogo informacija o sudarima, kao na primjer:

- Kada sudar počinje i završava
- Koje točke učvršćenja sudar dodiruje
- Vektor normale između dva učvršćenja
- Koliko je energije bilo uključeno u sudaru

Sve informacije o sudaru nalaze se unutar `b2Contact` objekta. Iz ovog objekta moguće je provjeriti koja se dva učvršćenja sudaraju te saznati položaj i smjer reakcije sudara. Postoje dva načina s kojima možemo dobiti `b2Contact` objekte iz Box2D-a. Jedan od načina je pregledavanjem kontakata za svako pojedino tijelo, a drugi način je korištenjem 'slušatelja' kontakata (eng. contact listeners). Ako imamo svijet u kojem se javljaju mnogi sudari postupak pregledavanja kontaka za svako pojedino tijelo postaje ne učinkovito. Postavljanje 'slušača' kontakata nam omogućava da primimo informaciju o tome kada se nešto 'zanimljivo' dogodilo sa našim tijelima. Pošto ćemo imati mnogo tijela za pregledavati u našem ćemo primjeru koristit metodu 'slušača' kontakata. [19]

'Slušač' kontakata je klasa (`b2ContactListener`) s četiri funkcije koje korisnik sam mijenja prema vlastitoj potrebi. Na kodu 3.23 moguće je vidjeti klasu `myListener` koja nasljeđuje klasu `b2ContactListener` te definirane metode koje ćemo koristiti za otkrivanje sudara između naših dva tijela.

```
class myListener : public b2ContactListener{
    public:
        void BeginContact(b2Contact* contact);
        void EndContact(b2Contact* contact);
        void PostSolve(b2Contact* contact, const b2ContactImpulse*
impulse);
        void PreSolve(b2Contact* contact, const b2Manifold*
oldManifold);
};
```

Kod 3.23: Dio koda koji sadrži definiciju klase myListener koja se nalazi unutar direktorija external/particle.h.

Metodu koju ćemo mi koristiti u našem primjeru je `PreSolve()` metoda. Ona nam daje priliku da promijenimo svojstva kontakata prije nego što dobijemo 'odgovor' sudara. Na sljedećoj stranici vidljiv je kod koji je korišten za implementaciju te metode.

```
      void myListener::PreSolve(b2Contact* contact, const b2Manifold*
oldManifold){
          b2Body* one=contact->GetFixtureA()->GetBody();
          b2Body* two=contact->GetFixtureB()->GetBody();
          int numPoints = contact->GetManifold()->pointCount;
          b2WorldManifold worldManifold;
          contact->GetWorldManifold( &worldManifold );
          int a= *((int*)one->GetUserData());
          if(a==0){

          }else if(a==2){
              for (int i = 0; i < numPoints; i++){
                  two->ApplyAngularImpulse( 20 );
                  two->ApplyLinearImpulse(b2Vec2(0,-50),two-
>GetWorldPoint( b2Vec2(worldManifold.points[i].x,
worldManifold.points[i].y )));
              }
          }
          int b= *((int*)two->GetUserData());
          if(b==0){

          }else if(b==2){

              for (int i = 0; i < numPoints; i++){
                  one->ApplyAngularImpulse( 20 );
                  one->ApplyLinearImpulse(b2Vec2(0,-50),two-
>GetWorldPoint( b2Vec2(worldManifold.points[i].x,
worldManifold.points[i].y )));
              }
          }

      }
```

Kod 3.24: Dio koda koji sadrži definiciju klase metode PreSolve() koja se nalazi unutar direktorija external/particle.cpp.

Najosnovniji dio informacije unutar sudara je informacija o tome koja se dva učvršćenja sudaraju. Ta se informacija dobiva naredbama `contact->GetFixtureA()->GetBody()` i `contact->GetFixtureB()->GetBody()`. Kako bi saznali u koliko se točaka događa sudar koristimo naredbu `contact->GetManifold()->pointCount`. 'Razvodnik' (eng. manifold) je samo 'fancy' naziv za liniju koja najbolje razdvaja dva učvršćenja. Nakon toga pregledat ćemo koja se tijela sudaraju, a u našem slučaju će biti bitan sudar između sustava čestica i velike čestice. Naredbe uz pomoć kojih pregledavamo sudare ova dva tijela jesu `one->GetUserData()` za prvo tijelo i `two->GetUserData()` za drugo tijelo. Uz pomoć varijable `a` moći ćemo zaključiti o kakvoj se čestici radi. Ukoliko se bude radilo o sudaru sa velikom česticom primijenit ćemo silu koja će prvo zavrtjeti česticu (`ApplyAngularImpulse()`), a zatim silu koja će našu česticu izbaciti u zrak (`ApplyLinearImpulse()`). [19]

Prilikom izrade objekta velike čestice koji će biti statički (neće se micati) koristit ćemo izmijenjenu metodu `createParticle()` unutar nove `Circle` klase.

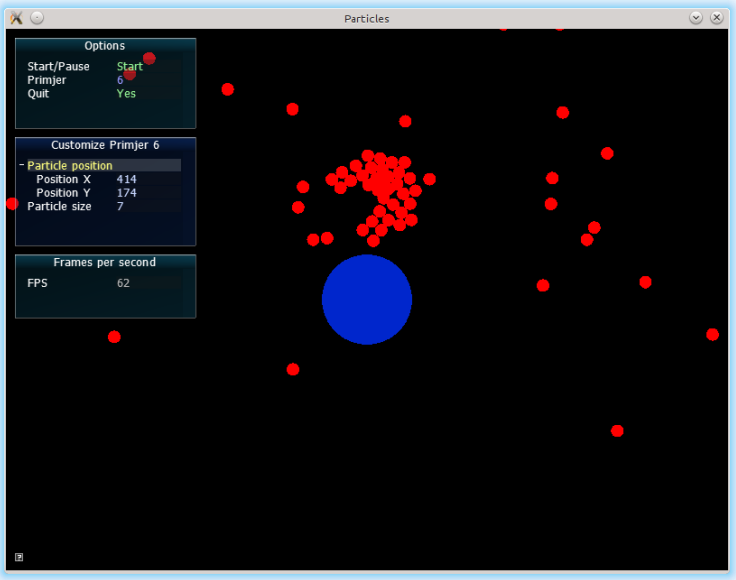
```
      void Circle::createParticle(int pos_x, int pos_y, int size,
b2BodyType type){
          p_alpha = 1.0;
          p_pos.x = pos_x;
          p_pos.y = pos_y;
          p_size = size;

          b2BodyDef myBodyDef;
          myBodyDef.type = type;
          myBodyDef.position.Set(p_pos.x*P2M,p_pos.y*P2M);
          p_body = getWorld()->CreateBody(&myBodyDef);

          b2CircleShape circleShape;
          circleShape.m_p.Set(0, 0);
          circleShape.m_radius = p_radius*P2M;

          b2FixtureDef myFixtureDef;
          myFixtureDef.shape = &circleShape;
          myFixtureDef.density = 0.1;
          myFixtureDef.friction = 0.0;
          myFixtureDef.restitution = 0.0;
          p_body->CreateFixture(&myFixtureDef);
          p_body->SetUserData(&p_num);
      }
```

Kod 3.25: Dio koda koji sadrži definiciju metode createParticle klase Circle, a nalazi se u direktoriju external/particle.cpp.

Kod unutar display() funkcije nećemo prikazivati pošto se bitno ne razlikuje u odnosu na već prikazane kodove. Sljedeći kod prikazuje kod primjera 6 koji se nalazi unutar main.cpp-a.

```
      if(TwGetBarByName("Customize Primjer 6") == NULL){
            TWbar6();
            circle[0].createParticle(400,300,50,b2_staticBody);
      }
      if(pause!=false){
            if(SDL_GetTicks()-old>=1.0){
              LastUsedParticle = unusedParticle();
              p[LastUsedParticle].createParticle(life,posX,posY,radius);

              for(int i=0; i<MAX; i++){
                  if(p[i].isDead()==false){
                        p[i].updateParticle(1);
                  }
              }
              old=SDL_GetTicks();
            }
      }
```

Kod 3.26: Dio koda koji sadrži naredbe za stvaranje dva različita sustava čestica, a nalazi se unutar main.cpp-a.



Slika 3.6: Prikaz odbijanja sustava čestica u odnosu na drugo tijelo (primjer 6) iz programa.

3.8. Stvaranje oblaka uz pomoć čestičnog sustava koji koristi teksture (primjer 7)

Pošto smo način stvaranja tekstura već ranije objasnili ovdje ćemo objasniti kako ih crtamo te način na koji smo implementirali simulaciju oblaka. Ova jednostavna simulacija oblaka stvorena je s idejom da se čestice povećavaju svakih šest milisekundi za 0.3 (minimalna veličina im je 10, a maximalna 40) i povećavaju vlastitu prozirnost za 0.01 dok ne postanu prozirne, odnosno nestanu. Funkcija `randFromInt()` je korištena s ciljem generiranja slučajnih koordinata sustava čestica oblaka. Jedina bitna razlika u odnosu na ostale primjere su dvije linije koda koje se nalazi u kodu 3.27. Prva linija koda koristi funkciju `LoadTexture()` koju smo već prije spomenuli, a služi za učitavanje slika i pretvaranje slika u teksturu. Druga linija koda koristi funkciju `DrawTexture()`, a služi za crtanje tekstura na tijelima odnosno česticama.

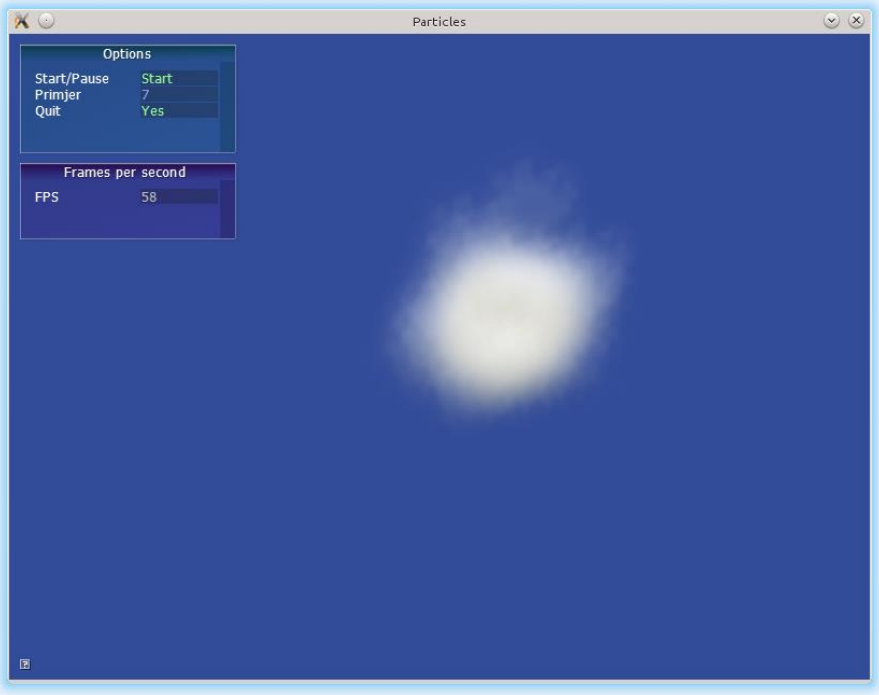
```
.
.
myglu=LoadTexture("data/cloud.png",&textw,&texth);
.
.
DrawTexture(position.x*M2P-5,position.y*M2P-5,myglu,size*4.0,size*4.0);
.
.
```

Kod 3.27: Dio koda koji sadrži funkcije za stvaranje teksture, a nalazi se unutar main.cpp-a.

Na sljedećem kodu vidljiva je implementacija funkcije `DrawTexture()`. Pošto smo većinu njenih naredba već ranije opisali ovdje ćemo opisati samo naredbe koje se do sada nisu pojavljivale. Naredbu `glTexCoord2i()` koristimo prilikom označavanja točaka koordinata tekstura, a naredbom `glVertex3f()` pridružujemo dužinu i širinu teksturama.

```
      void DrawTexture(int x, int y, GLuint textureid,int textw,int
texth) {
               glBindTexture(GL_TEXTURE_2D, textureid);
               glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
               glEnable(GL_TEXTURE_2D);
               glBegin(GL_QUADS);
               // gornji lijevi vrh
               glTexCoord2i(0, 0);
               glVertex3f(x, y, 0);
               // gornji desni vrh
               glTexCoord2i(1, 0);
               glVertex3f(x+textw, y, 0);
               // donji desni vrh
               glTexCoord2i(1, 1);
               glVertex3f(x+textw, y+texth, 0);
               // donji lijevi vrh
               glTexCoord2i(0, 1);
               glVertex3f(x, y+texth, 0);
               glEnd();
               glDisable(GL_TEXTURE_2D );
      }
```

Kod 3.28: Dio koda koji sadrži funkcije za stvaranje teksture, a nalazi se unutar main.cpp-a.



Slika 3.7: Prikaz stvaranja oblaka uz pomoć čestičnog sustava (primjer 7) iz programa.

4. Brzina renderiranja

Još u samim početcima ovog rada spomenuli smo i opisali što je to renderiranje, a u ovom poglavlju bavit ćemo se pitanjem kako mjeriti brzinu renderiranja. Većina ljudi danas renderiranje mjeri pomoću FPS-ova (eng. Frames Per Second), no FPS-ovi nisu uvijek najbolji način za mjerenje performansi jer nisu linearni u vremenskoj domeni. Bolje je izmjeriti takozvani vremenski okvir (.eng frame time) koji je inverzija FPS-ova. Vremenski okvir izračunati ćemo tako da broj jedan podijelimo sa brojem FPS-ova. Dakle ako su FPS-ovi jednaki 25 vremenski će okvir iznosi 1/25 = 0.04 sekundi.

Problem FPS-ova objasniti ćemo u sljedećem primjeru. Recimo da renderiramo brzinom od 180 FPS-ova. Sada ćemo dodati još par čestica za renderiranje da se naši FPS-ovi smanje na 160. Koliko je zapravo to loše? Izgubili smo 20 FPS-ova, ali koliko je sada sekundi više potrebno prilikom renderiranja? Jednostavnim izračunom dolazimo do 1/180 - 1/160 = 0.00069 sekundi. Pretpostavimo sada da su naši FPS-ovi jednaki 60 i dodajemo čestice za renderiranje dok FPS-ovi ne padnu na 40. Koliko će se vrijeme renderiranja produžiti u ovom primjeru? Sličnim izračunom kao i prije dolazimo do 1/60 - 1/40 = 0.00833 sekundi. Iako su FPS-ovi u oba slučaja pali za 20, usporavanje u prvom slučaju nije jako veliko, ali je zato u kasnijem slučaju osjetno veći.

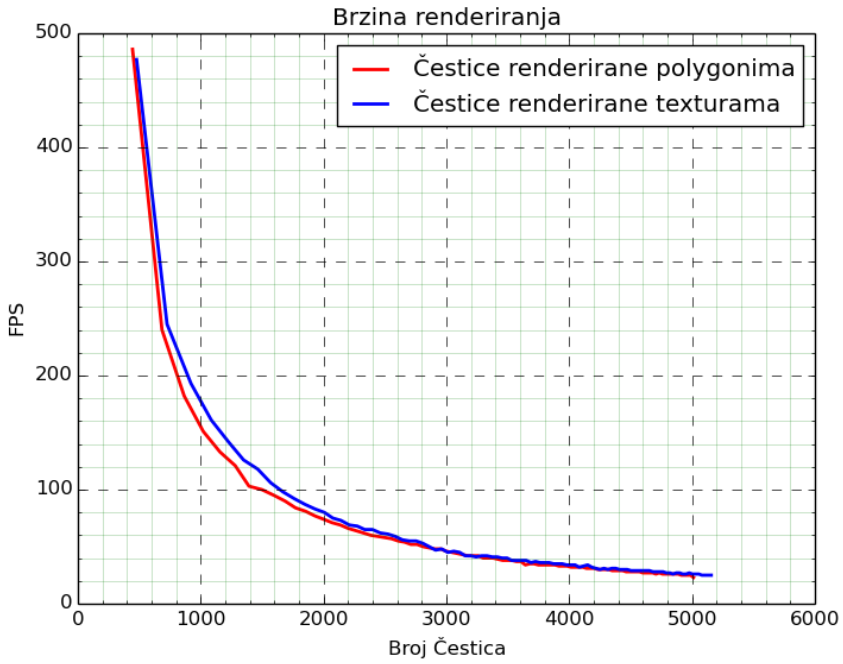
Za mjerenje brzine renderiranja u ovom primjeru koristit ćemo se FPS-ovima na temelju kojih ćemo kasnije izračunati vremenski okvir. Prilikom mjerenja FPS-ova koristit ćemo sljedeći kod (4.1). U ovom kodu treba napomenuti da se varijabla `fps_frames` povećava unutar display funkcije što nije vidljivo ovim kodom.

```
if (fps_lasttime < SDL_GetTicks() - 1000)
   {
      fps_lasttime = SDL_GetTicks();
      fps_current = fps_frames;
      fps_frames = 0;
   }
```

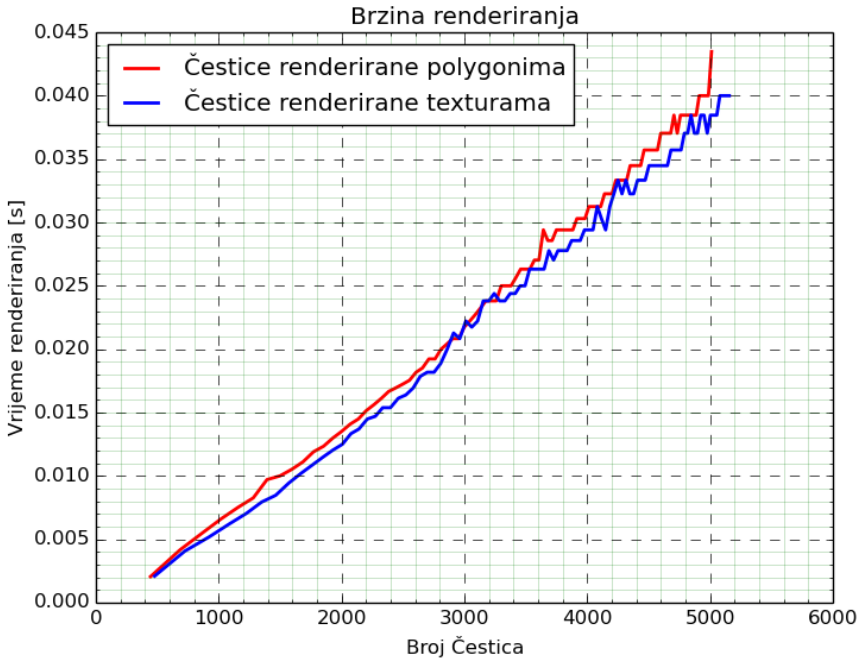
Kod 4.1: Dio koda koji sadrži funkcije za mjerenje FPS-ova, a nalazi se unutar main.cpp-a.

Iako ovo nije baš najispravniji način za mjerenje performansi našeg programa, nama će biti dovoljno dobar za dobiti dojam o tome kako količina naših čestica djeluje na vrijeme renderiranja.

Prilikom provedenih testiranja programa dobiveni su sljedeći rezultati renderiranja.



Slika 4.1: Prikaz grafa brzine renderiranja na osnovu broja čestica i FPS-ova.



Slika 4.2: Prikaz grafa brzine renderiranja na osnovu broja čestica i potrebnog vremena renderiranja.

Računalo:	HP Pavilion dv6 Notebook PC (A2X59EA#ABZ)
Ram memorija:	2*4GiB SODIMM DDR3 Synchronous 1333 MHz (0,8 ns)
Procesor:	Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz
Grafička kartica:	Radeon HD 6770M
Disk:	250GB Samsung SSD 840

Tablica 4.1: Prikaz korištene konfiguracije prilikom renderiranja.

Iz dobivenih rezultata zaključujemo da renderiranje poligona čestice zahtjeva više procesorskog vremena u odnosu na renderiranje same teksture čestice. Na slici 4.1 moguće je vidjeti odnos između broja čestica i FPS-ova programa. Iz slike je moguće zaključiti da za održavanje željenih 60 FPS-ova ne smijemo stvarati više od 2400 poligonalnih čestica odnosno 2550 čestica sa teksturom. Kako dobiveni brojevi (2400 i 2550) nisu veliki brojevi za sustave čestica (imamo sustave čestica koji sadrže po par milijuna čestica) zaključujemo kako bi se ovaj program moglo još dodatno optimizirati i poboljšati.

5. Zaključak

Korištenjem programskog jezika C++ te knjižnica OpenGL, Box2d, SDL i AntTweakBar stvoren je program koji korisniku pruža GUI s mogućnošću prikaza različitih čestičnih modela, od onih najjednostavnijih do onih složenijih. Program prikazuje i različita svojstva i mogućnosti koje pruža fizička knjižnica Box2D. Korisniku se putem GUI-a daje velika mogućnost prilagođavanja čestičnih sustava poput promjene boje, koordinata stvaranja sustava, veličine pojedinih čestica i slično. Pojedini primjeri ovog završnog rada služe korisniku da postepeno shvati i nauči implementirati čestične sustave. Sve počinje sa prvim primjerom u kojem se stvara jedna čestica, zatim se nakon toga stvara sustav čestica, pa više sustava čestica sve dok ne dođemo do sustava čestica koji koristi teksture. Svaki sustav čestica koristi fizičku implementaciju putem Box2D-a, što dodatno omogućava korisniku shvaćanje ove fizičke knjižnice. Sustavi čestica se većinom koriste u simulacijama i računalnih igricama, upravo nam ti sustavi omogućavaju simulaciju i prikaz vatre, oblaka, dima, kiše, snijega, itd.

Literatura

[1] https://en.wikipedia.org/wiki/Computer_graphics_(computer_science)

[2] http://ahyco.uniri.hr/Seminari2010/Racunalna_grafika/images/VectorBitmapExample.png

[3] https://irislovatic.files.wordpress.com/2013/11/dyer_et_al_sgp07.jpg

[4] https://en.wikipedia.org/wiki/Animation

[5] https://en.wikipedia.org/wiki/Rendering_(computer_graphics)#/media/File:Render_Types.png

[6] https://bs.wikipedia.org/wiki/Datoteka:Globe_and_high_court.jpg

[7] https://bs.wikipedia.org/wiki/Datoteka:Globe_and_high_court_fix.jpg

[8] http://www.cplusplus.com/info/description/

[9] http://www.cmake.org/overview/

[10] https://www.opengl.org/wiki/Main_Page

[11] https://www.libsdl.org/

[12] http://www.zemris.fer.hr/predmeti/irg/Zavrsni/08Piljek/index.html

[13] http://www.zemris.fer.hr/predmeti/irg/Zavrsni/08Piljek/ch3.html

[14] https://www.libsdl.org/projects/SDL_image/

[15] http://box2d.org/about/

[16] http://box2d.org/manual.pdf

[17] http://anttweakbar.sourceforge.net/doc/

[18] http://natureofcode.com/book/chapter-4-particle-systems/

[19] http://www.iforce2d.net/b2dtut/collision-anatomy

Popis slika, kodova i tablica

Prilozi

Izvorni kod programa pisanog u C++ programskom jeziku, potrebne knjižnice za pokretanje programa, CMake datoteka za kompajliranje, te python skripta za renderiranje nalaze se na optičkom mediju priloženom uz završni rad.